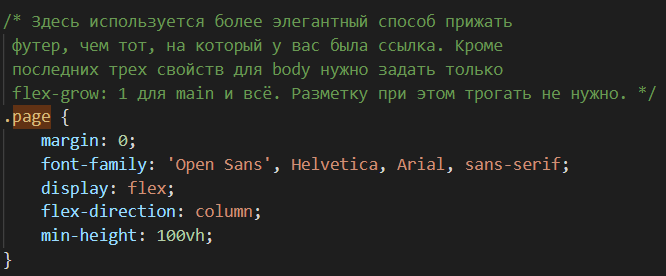
Модуль А4.

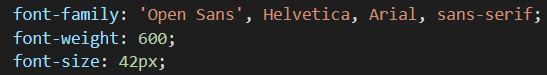
Он посвящен декоративному оформлению. Добавление изображений фона, декоративные элементы css. Создание интерактивных форм для отправки данных, фильтров и тд.

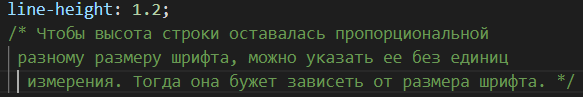
Начало изучение **javascipt**, консоль браузера и первый js-script.

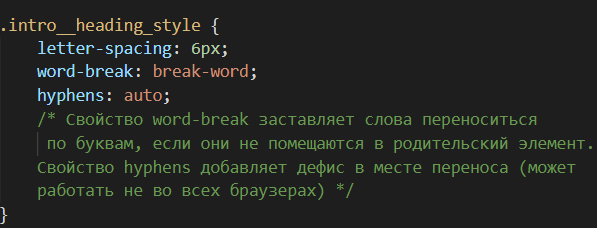
Кое-что по комментариям из их варианта.

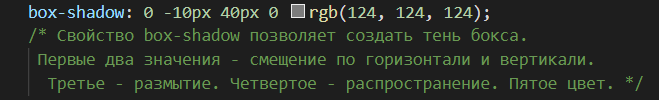


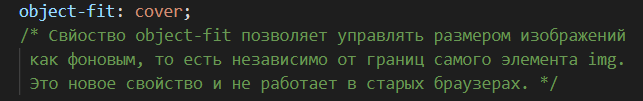


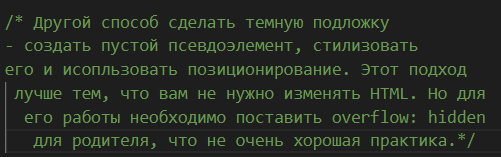














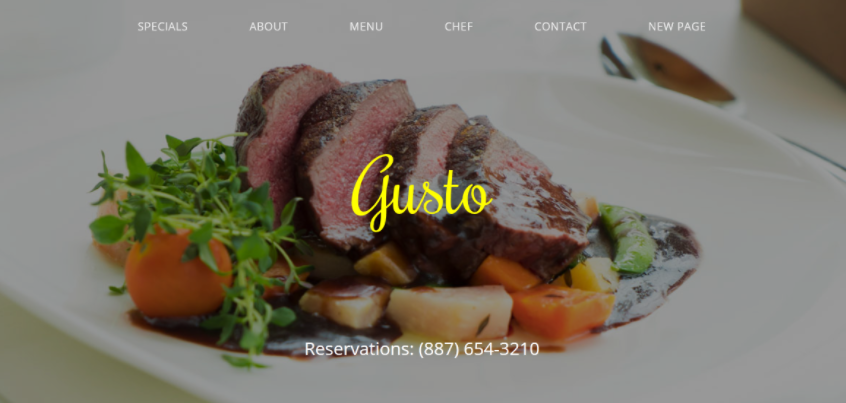
**Фоновые изображения**

Визуальное оформление — это важная часть сайта.

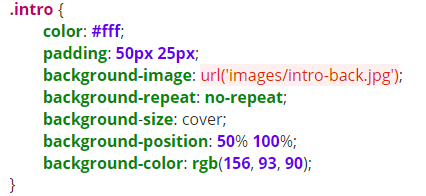
В css есть вид визуального оформления – **фоновые изображение.**

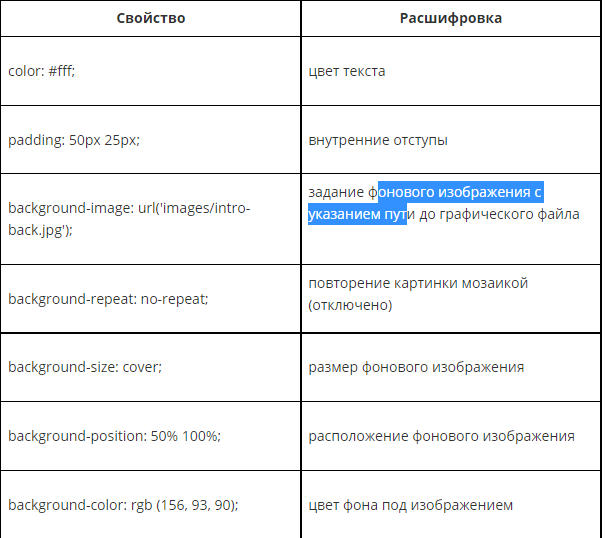
Не путать с изображениями, которые мы добавляем в html. В html содержимое, в css оформление.

Вот пример css фонового изображения:



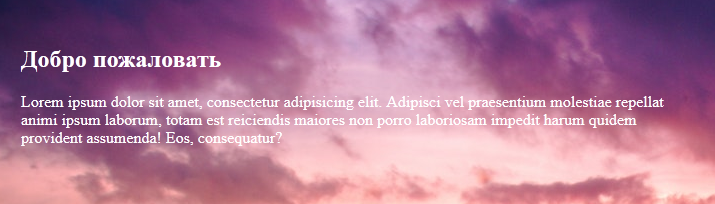
Вот пример кода этого изображения





Задать изображение можно любому элементу.

Если изображение достаточно большое, то оно заполнит весь элемент.

 Однако если это не так, то оно по умолчанию включит повторение



Это не плохо если фон — это орнамент. Иначе это выглядит не очень красиво. Но повторение можно отключить. Свойством **background-repeat: no-repeat;**



Чтобы изображение повторялось только по горизонтали используют значения **repeat-x,** а для вертикали **repeat-y.**

Также есть возможность управлять размерами картинки с помощью свойства **background-size.**

Есть несколько свойств:

**cover –** изображение заполняет все свободное место, пропорции сохраняются, а лишняя часть отрезается.

**contain -**  Изображение масштабируется так, чтобы оно заполняло максимальное пространство, при этом показывалось полностью.

Кроме того размер можно указывать в процентах.



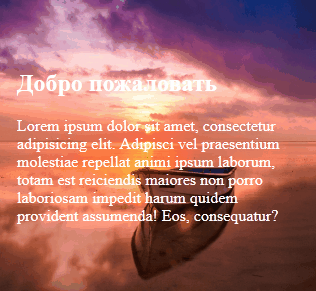
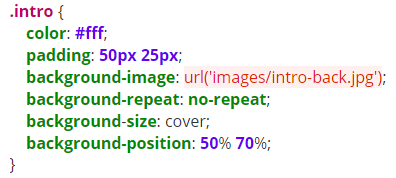
70% от родителя.

Можно указать сразу два свойства, тогда первое значение будет означать процент по вертикали, а второе по горизонтали:

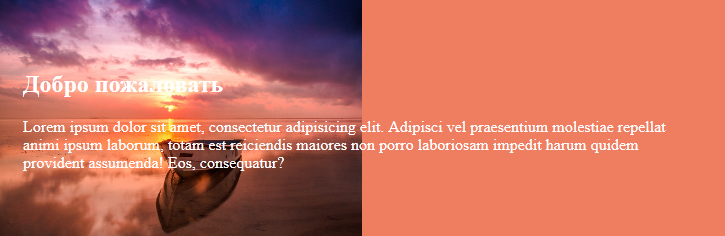
**background-size: 90% 80%;**

Фон можно двигать относительно элемента по горизонтали и по вертикали свойством, **background-positon.** Это полезно, если необходимо, чтобы важный объект всегда был виден на разной ширине вьюпорта. Например, располагался всегда по центу по горизонтали, но внизу по вертикали. Особенно полезно при резиновой верстке. Пример:



Когда вы используете картинку в качестве фона, можно задать фоновой цвет. Он будет находится под изображением, это значит, что если картинка не прозрачная и занимает все пространство, то под ней не будет видно цвета. Но если изменить размер фоновой картинки на 50%, то цвет фона будет виден.

В данном примере цвет фона это средний цвет изображения. Узнать средний цвет изображения можно через сервис <https://matkl.github.io/average-color/> или в photoshop 

## Псевдоэлементы

Иногда нужно добавить рядом с каким-то элементом небольшое декоративное изображение, иконку, например. Для этого хорошо подходят псевдоэлементы(не псевдоклассы).

Два самых популярных - **::before** и **::after**. Они позволяют добавить какое-то декоративное содержимое перед или после элемента.

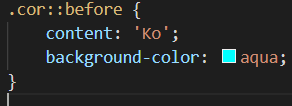
Псевдоэлемент подписывается после содержимого через двоеточия.



Главное свойство для псевдо-элементов – **content.** В нем можно задать текст или любые юниод-символы, разумеется в кавычках.

**Простой пример:**





**Или after**





Можно использовать таким же образом Unicode, для этого нужно в кавычки записать ту часть, которая идет после плюса c поставленной перед кодом косой чертой.

, в данном случае **\00A9**

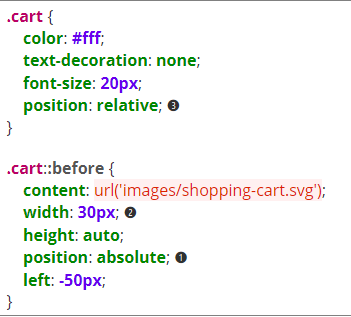


Можно также вставить и путь к изображению.



**Важно:** внутри **content** писать **html** код **нельзя.**

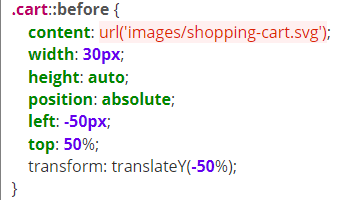
Если предположить, что мы хотим изображение не сверху, а слева. То стоит воспользоваться абсолютным позиционированием.

 Важно не забыть задать позиционирование самому элементу, относительно которого будет движение.



А чтобы выровнять абсолютно с позиционированный элемент точно по центру, можно использовать свойство **transform: translate.** Оно позволяет **сдвинуть элемент относительно собственной ширины.**  Сдвигать можно как по горизонтали, так и по вертикали: **translateX** и **translateY.** Или одновременно **translate.**

Вот пример сдвига в половину собственной ширины наверх:



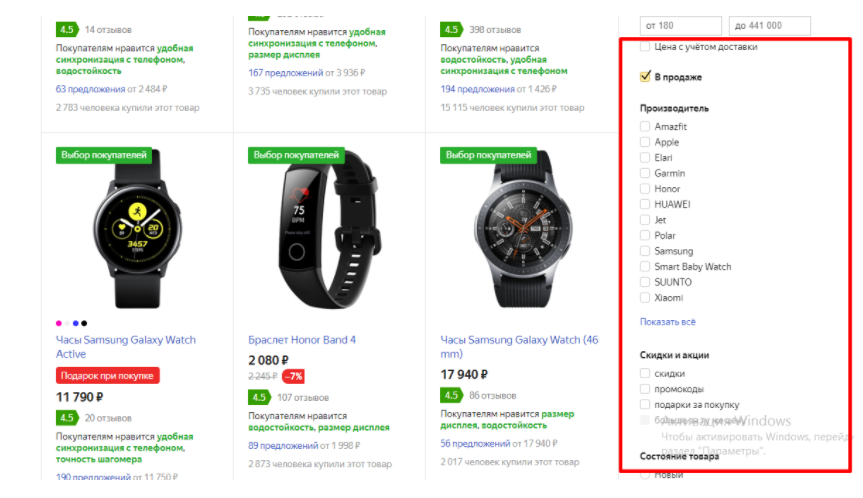
Псевдоэлементы используются только для **декораций**, которые можно убрать без потери смысла, например, **пункты списка, линии и полоски,** создание элементов с нуля - <https://css-tricks.com/the-shapes-of-css/>

## Формы отправки данных. Основы форм

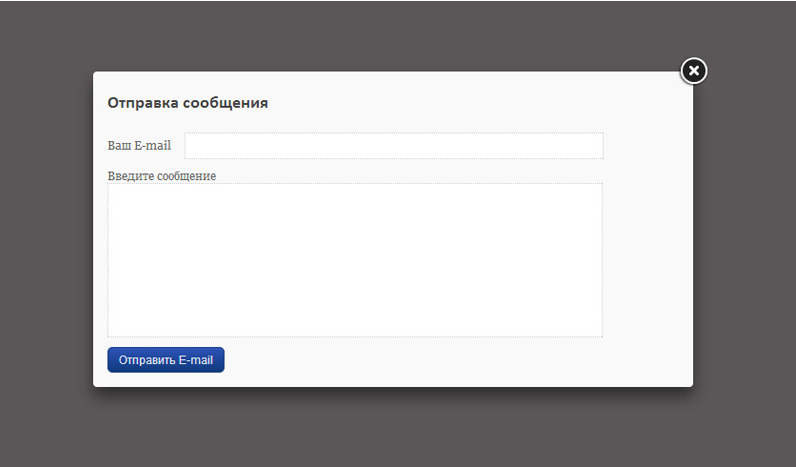
Они нужны, чтобы пользователи сайта могли вводить и отправлять на сервер различные данные.

Например, оставлять отзывы, использовать фильтры в онлайн магазине. Входить под своей учетной записью, поиск по сайту и т.д.

Воn пример фильтра в Яндекс.Маркет:



Форма отправки данных.

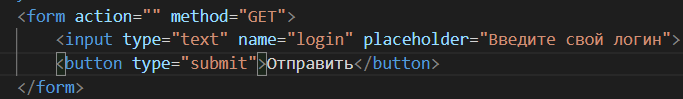


Поиск по сайту



Форма содержит различные интерактивные элементы, с помощью которых пользователь может ввести текст, выбрать опции и отправить это все на сервер нажатием кнопки.

Вот html код самой обычной формы с кнопкой.



Вот как она выглядит



А теперь разберем ее код.

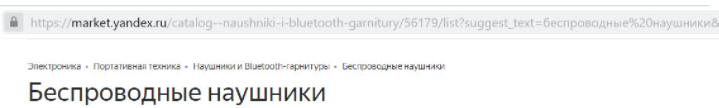
Первым идет тег <**form>**. Внутри него уже и будут находится поля и кнопки. Как понять, что кнопку или поле нужно занести в **<form>**? Если все данные должны отправляться вместе, то их нужно поместить в одну форму. Это будет нагляднее, позже с более сложными формами.

У элемента **form** есть два необходимых атрибута. **Action** и **method.** Атрибут **action** указывает адрес программы на сервере, которая будет получать и обрабатывать отправленные данные. Пока этот атрибут пустой, но позже будет несколько способов протестировать форму.

Атрибут **method –** сообщает, какой метод отправки данных использовать. Пока разберемся с методом **GET.**

**GET** отправляет данные внутри URL адреса, добавляя к нему так называемые **GET-параметры.** Так сервер получает данные из URL. Пользователь сможет увидеть эти данные прямо в адресной строке браузера.

Когда пользователь, скажем ставит какую-то галочку в браузере, эта информация добавляется в адресную, строку



Это очень удобно ведь в адресной строке можно так сохранить не просто страницу, а страницу

с сохраненным списком товаров к примеру. Основной элемент форм – **input,** то есть элемент ввода данных. “Инпуты” бывают самыми разными, их тип указывается в атрибуте **type.**

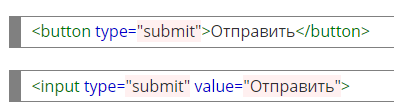
Значение атрибута **type=”text”** означает, что этот элемент является текстовым полем и предназначен для ввода текста. Он универсален для любых текстовых полей. Есть и более узконаправленный, например, для паролей или почты.

Типы инпутов можно посмотреть здесь: <https://developer.mozilla.org/ru/docs/Web/HTML/Element/input#attr-type> –

Чтобы сервер понял, что представляют собой полученные данные, инпутам нужно задать имя в атрибуте **name.** Оно может быть любым, как и имя класса. Но, чтобы не путаться следует давать понятное имя, пример: **name=”login”**

атрибут **placeholder** подсказывает пользователю, что в него должно вводится. Как только начинает ввод, этот текст исчезнет.

Для отправки данных нужна кнопка. Она может быть сделана в виде элемента **button** или **input.** Но атрибут у **type** у них должен иметь значение **sumbit.**



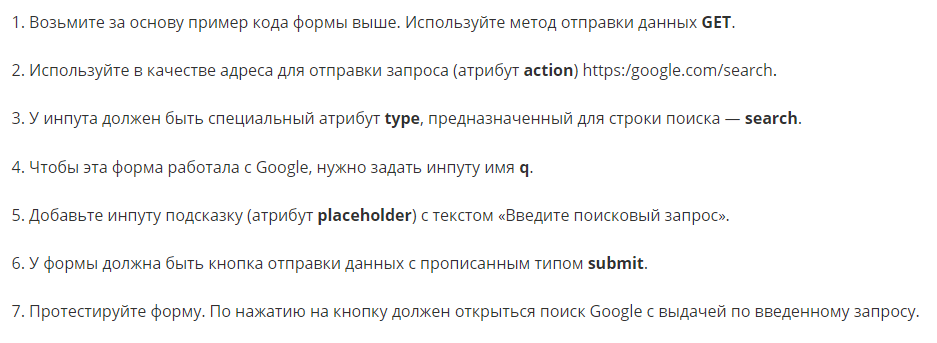
**Разница в том, что внутри тега button можно поместить и другие элементы, форматированный текст или изображения. А внутри input можно вставить только текст.**

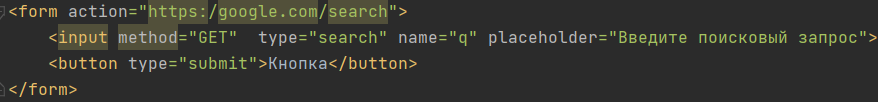
Если кнопка находится внутри формы, то есть внутри двух тегов **<form></form>**

То она автоматически будет отправлять данные этой формы. Кроме нажатия на кнопку пользователь может также отправить данные клавишей **Enter.**

По умолчанию у **button** стоит тип отправки данных, но на всякий случай его все же стоит прописывать. **type=”sumbit”**

**Задание**





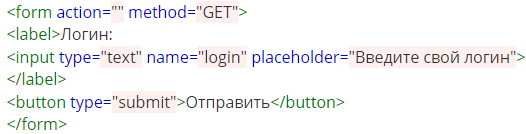
По ошибке сначала записывал атрибут **action** в **input.** Из-за этого ясное дело не работало.

## Подписи к полям ввода

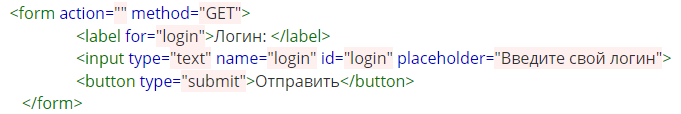
Помимо подписей, которые являются содержимым полей формы, этим полям можно задавать дополнительные подписи. Реализовать это можно с помощью тега **<label>**. Так можно обозначить за что, то или иное поле отвечает.

Связать **label** с полем можно двумя способами.

1. Обернуть поле в **label.**



1. Связать их через атрибут **for** у **label** и **id** у **input.**

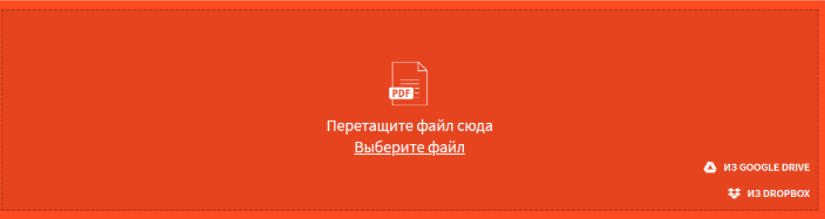


Хотя в данном примере **name** и **login** совпали, но это не обязательно. Напомню, атрибут **name** для сервера, чтобы проще понять, что содержит поле.

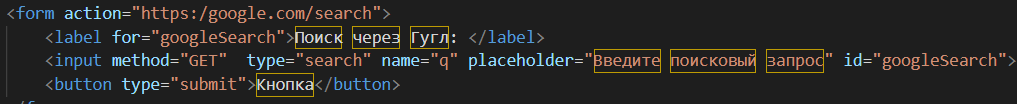
Из других преимуществ **label,** в отличии от тех же параграфов, связанный **label** при клике на него будет подсвечивать поле, с которым он связан. Если это чекбокс, то вместо самого маленького квадрата с галочкой, можно нажать на саму надпись и чекбокс сработает.

Также **label** удобен для дизайна. Не все элементы также легко стилизовать, как **div** или **p.** Поэтому верстальщики могут, например, скрыть сами поля, а вместо них стилизовать **label.** Так к примеру делают стилизованные чекбоксы.

Пример стилизованного поля ввода для загрузки файлов:

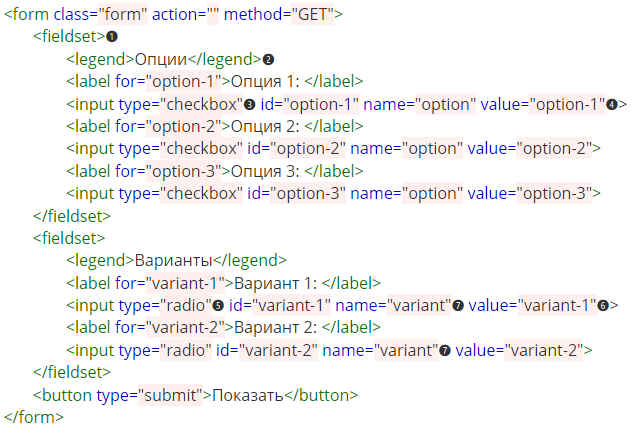


**Дополнение к написанному поля Google поиска**



## Другие виды инпутов

Инпутов довольно много. Многие для своих узких целей. Рассмотрим подробнее.



Поля одного вида в этой форме объединены через тег **fieldset.** Он служит для группирования элементов формы. У каждого **fieldset,** может быть заголовок созданный через **legend**. Он необязателен. Но сложную форму легче воспринимать с ним.

**input** с типом **checkbox,** представляет из себя поле, которое можно включить или выключить нажатие, можно выбрать сразу **несколько чекбоксов одновременно**.

У каждого **input** есть атрибут **value.** Он может по-разному функционировать на разных типах **input.**

Для инпутов в которых нет текста **value** содержит информацию, которую нужно передать на сервер – в данном случае текстовое обозначение варианта.

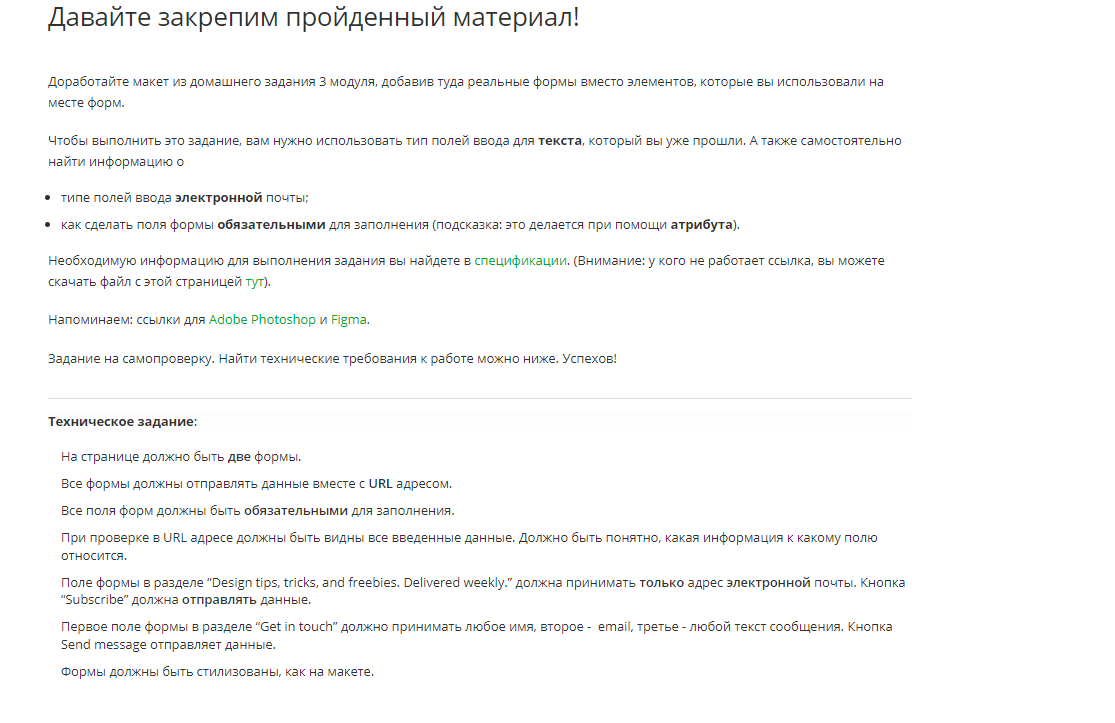
**В пункте 4** чекбокс имеет **value=”option-1”.** Эта информация отправится на сервер при выборе этого пункта. **Name** для этой цели не подходит, потому что у трех в этом контексте имя содержит одинаковое со всеми общее название **option** – он объединяет их по смыслу**.**

Другой похожий инпут это **радиокнопка(type=”radio”)** Схож с чекбоксом, но из группы таких кнопок может быть выбрана только одна. Также как и **checkbox** передают значения на сервер через **value.** Чтобы **радиокнопка** относилась к группе обязательным условием должно быть соблюдение одинаково имени **name.** Иначе она просто не будет переключаться.

Чтобы в зависимости от выбранных в форме элементов менялось и содержимое страницы информация с формы должна передаться на сервер, который эту информацию обработает и вернет из базы данных новую информацию, а затем браузер ее покажет.

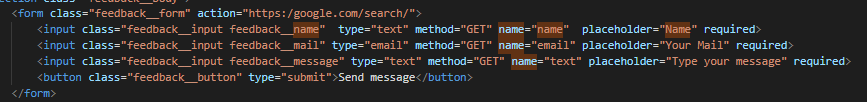
**Но можно менять содержимое и через чистый JS.**

**Задание “Формы макета”**



Код





Отправка





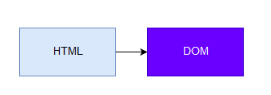
Отрисовка страницы и DOM

Начало работы с JS, хотя я уже осведомлен. Но сначала поймем как из кода получаются готовые сайты на экране пользователя.

Общая схема веб-проекта.

****

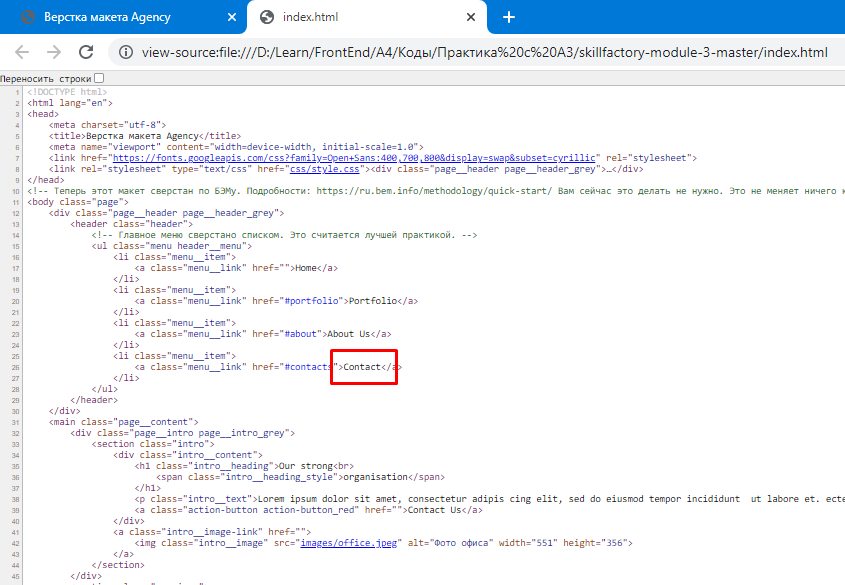
Каждая часть пишется на своем языке. По порядку. Браузер открывает **html** файл, затем читает и выполняет код. Этот процесс называется **парсингом(**А я думал парсинг это распределение чего-то**)**

****

Затем браузер создает из кода структуру, называющуюся, **DOM (**Document object model или объектная модель документа**).** Именно с ней происходит взаимодействие в инструментах разработчиков.

**DOM** состоит из узлов. Это древовидная структура. Как правило они совпадают с html элементами кода, но бывают исключения (псевдоэлементы, исправленные ошибки в кода)

**DOM** всегда виден в инструментах разработчика и в нем актуальный код с исправленными ошибка.



Это **html код** открытый в просмотре кода страницы, это не dom. В этом коде прописан **JS ,к**оторый должен заменить выделенную строку словом жопа.

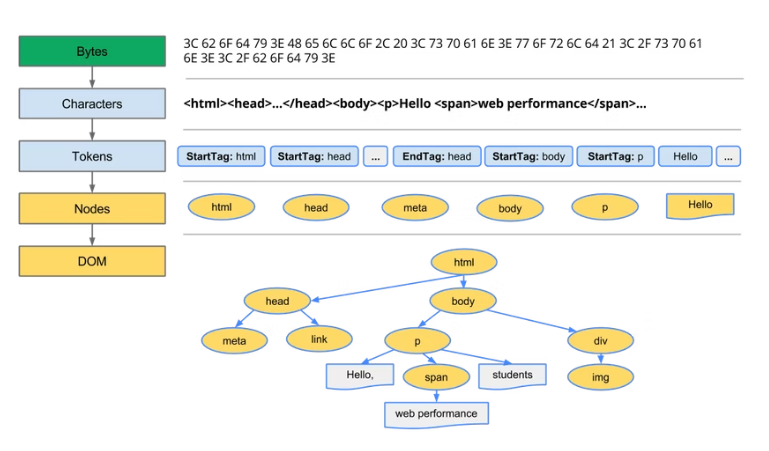


Вот это **DOM.** И **JS** способен его динамически изменять.

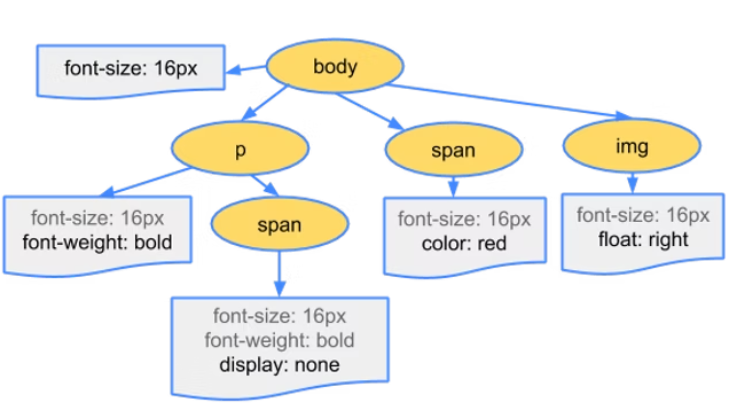
Используя **AJAX** и **шаблонизацию** можно заметить, что  **DOM** тоже будет заметно отличаться. Осоюенно это заметно на страницах, сформированных с помощью подключенных скриптов для первоначальной загрузки. Когда в JS мы пишем **document** мы обращаемся к **dom,** когда добавляем чему-то обработчик, аналогично.

Похоже получается и с **CSS.** Если **html** обращается в **Document object model,** то **css** преобразуется в **CSS Object model(CSSOM)**

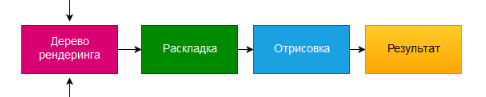
Вот подробное изображение преобразования на всех этапах.



И по умолчанию дети наследуют свойства своего родителя.



Все эти этапы происходят каждый раз, когда формируется html и чем он больше, тем дольше. **CSS** при формировании имеет такую же древовидную структуру. Затем он рекурсивно уточняет свойства от потомка к потомку.



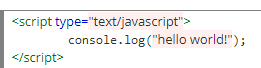
Медиа запросы вызывают изменение в геометрии элементов, то есть заставляют **DOM** переформироваться, что не лучшим образом сказывается на производительности.

**A4.9 Основы Javascript и подключение**

Зачем нужен JS. Основная цель это манипуляция содержимым веб-страницы. Можно как угодно влиять на DOM, узлы, содержимое, атрибуты, css.

JS можно подключить разными способами. То, как он подключается, может сильно повлиять на работоспособность всей страницы.

Он может быть написан внутри html



**type=”text/javascript” –** является частью стандарта, но сейчас его наличие не обязательно, современные браузеры и так распознаю и выполняют код.

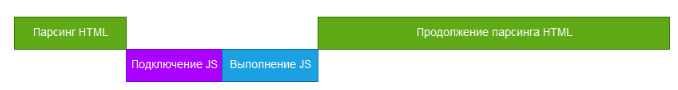
Второй способ как и **CSS** отдельным файлом.



Когда браузер натыкается на **JS** он сразу начинает его исполнение. Это может плохо сказаться на производительности. Поэтому код принято вставлять либо в конце. Либо параллельно с **html**, гдеэтого требует логика выполнения.

Зависит от ситуации.





Для подключения JS могут использовать еще 2 атрибута. Оба применяются без значений.

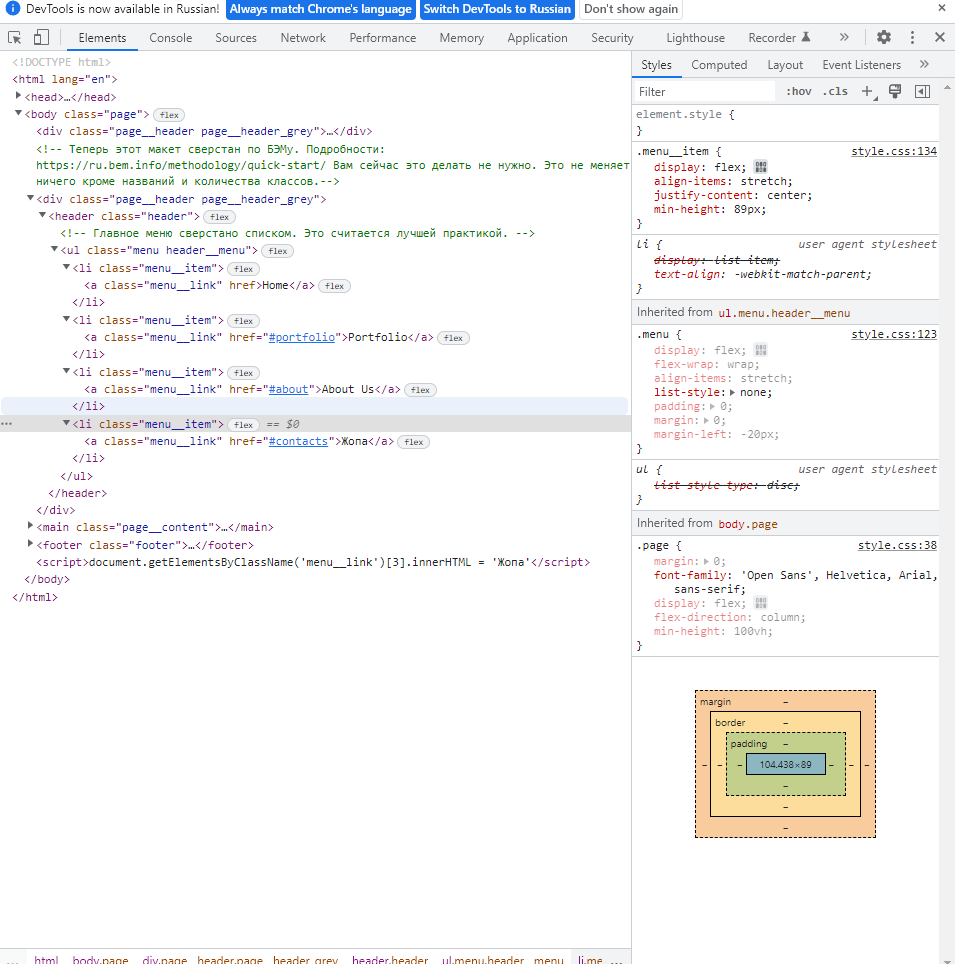
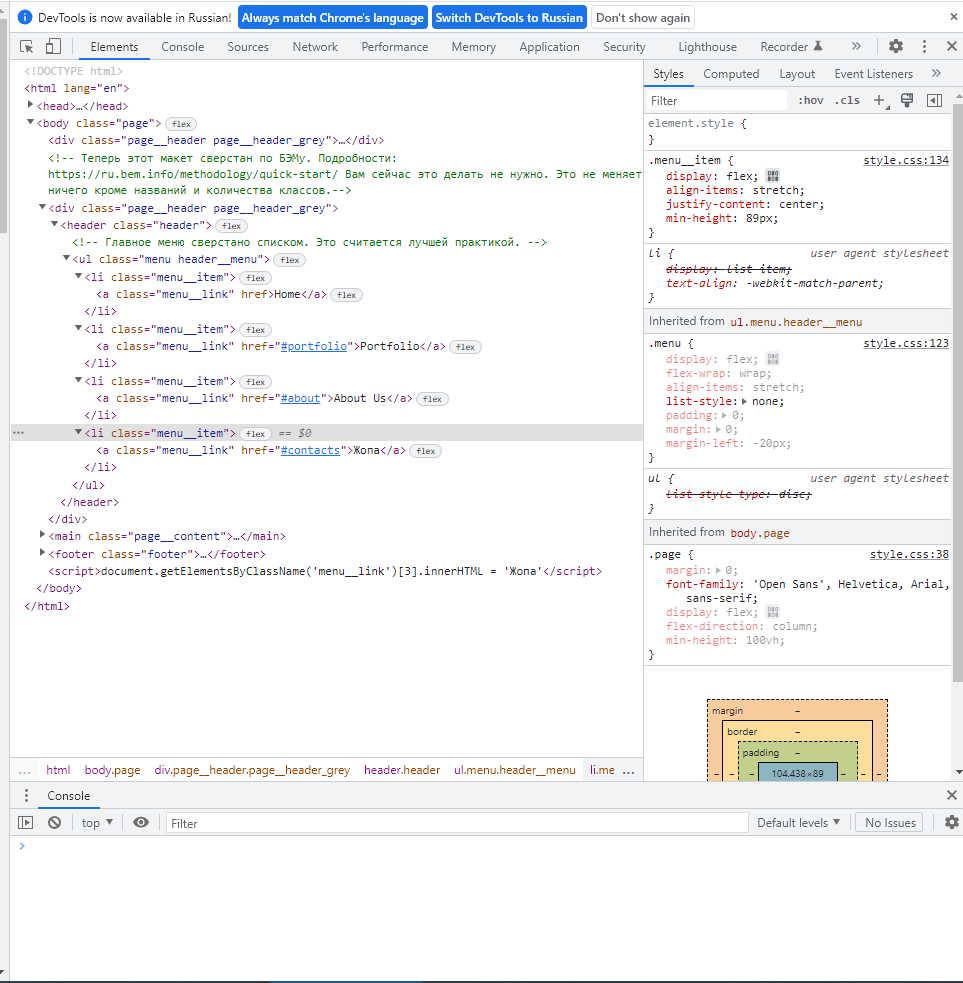
**Async –** логический атрибут. Указывает браузеру загружать скрипт асинхронно. Что значит что этот JS файл выполняться без ожидания загрузки и отображения страницы.



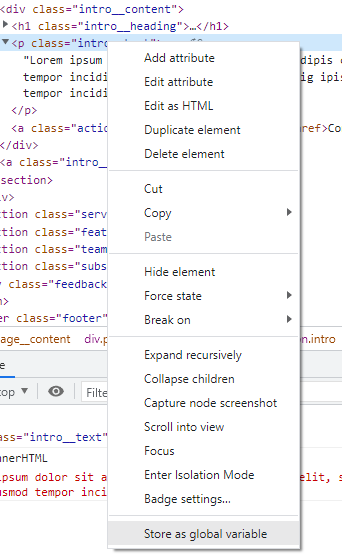
**Defer –** логический атрибут. Противоположность **async,** выполнение скрипта будет отложено, пока html страница не будет готово. Если написать и **defer**  и a**sync,** то второй игнорируется.

A4.10 Взаимодействие с DOM через консоль

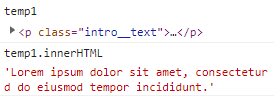
Видос. Да, когда в инструментах разработчика есть консоль. Но иногда нужно видеть одновременно и DOM и консоль. Тогда можно нажать на ESC и консоль появится прямо на вкладке элементов

Если нажать на элемент правой кнопкой мыши и выбрать нижний пункт **Store as global variable.** То можно добавить этот элемент как переменную в консоль. У нее даже сгенерируется имя.



**Имя переменной temp1.**



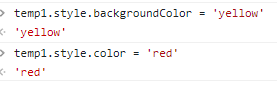
Допустим его можно удалить.



После этого действия он пропадет не только из страницы, но и из DOM. И это отличается от условного **display: none;** в **CSS**

В css он просто скрыт. А сейчас полностью удален из кода. **CSS** не может влиять на DOM. JS может.

Вот пример изменения CSS.



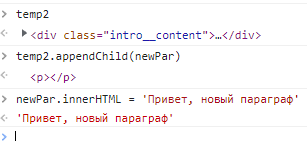


Чтобы что-то создать что-то в DOM. Нужно сначала создать узел. А затем добавить его куда-то в DOM.

Создадим переменную.

**let newPar = document.createElement('p');**

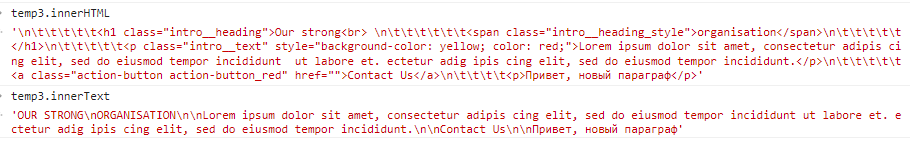
Методом **appendChild** можно добавить созданный элемент в качестве тега. Однако это тег пустой. С помощью метода **innerHTML** добавляем к нему текст. И этот текст отобразится внутри DOM.





Хм.. а вместо **innerHTML** можно было использовать **innerText.** А в чем разница?

В этом случае разницы нет. А так **innerHTML** возвращает в качестве строки весь html дочерних элементов. А **innerText** только текстовое содержимое дочерних элементов.



Понятно.

**A4.11. Основы синтаксиса JavaScript**

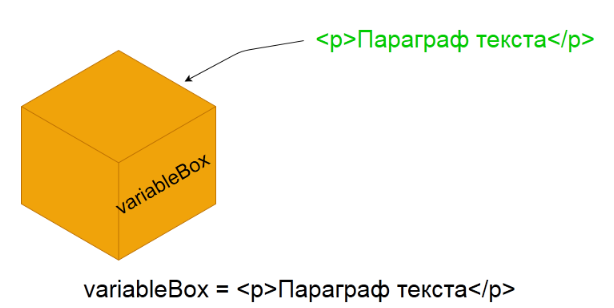
Я уже читал книгу по JS. Посмотрим выучу ли я еще что-то.



Понятное дело используется для отладки и на готовом сайте их принято скрывать.

Точка-запятая в конце каждой команды не обязательна. Но все же рекомендуют использовать ее.

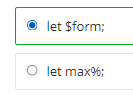
**Переменная –** это хранилище для различных типов данных. Строк, чисел, объектов.



Я понимаю, это абстракция. Но пример плохой.



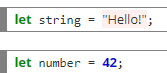
**Переменная не может начинаться с цифры.**

****

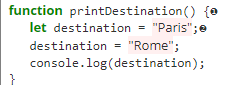
**Let max%** нельзя из-за спец.символа.

Процесс создание переменной называется объявлением.

Процесс задание переменной значения называется инициализацией.

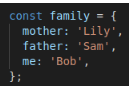


Функции. Ну описывать их точно не буду, хорош уже.

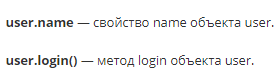


**Объекты, свойства, методы.**

Объект похож на переменную. Но хранит сложные типы данных. Вот пример.



Полезны, когда нужно записать информацию, представляющую из себя сложную структуру. Объекты обладают свойствами и методами.



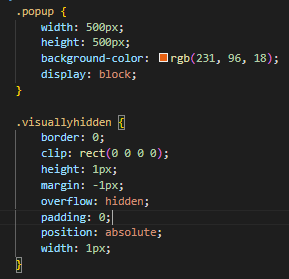
**Создание динамики**

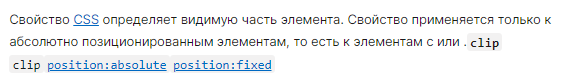


Метод **.toggle** добавляет или удаляет определенное свойство. В данном случае этот метод добавляет класс. ~~Но думаю также легко он может добавлять и убирать определенный css.~~  Не нельзя, **.toggle** в качестве параметров принимает только класс.

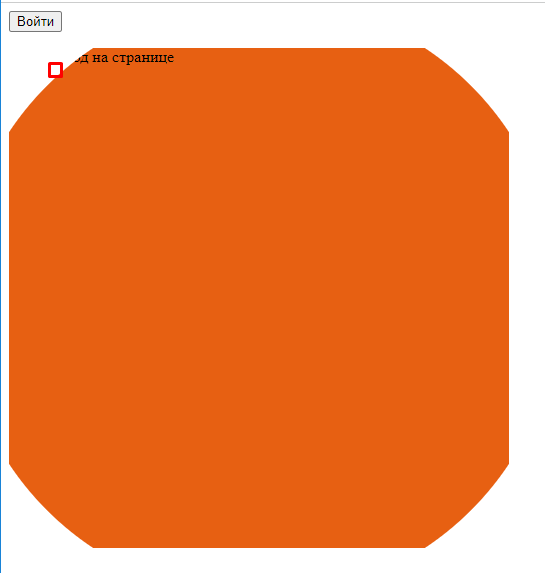
А так через метод **.addEventListener(),** выбранной кнопке добавилось определенное свойство.

В данном случае используется **display: none;** И если внутри блока был бы текста, то когда это поле скрыто, читалки и поисковики никак не смогут найти этот текст, пока он скрыт. Если же задача скрыть только визуально. То можно просто ужать этот блок до минимального размера и он никак не будет отобраться, хотя доступ к тексту останется.

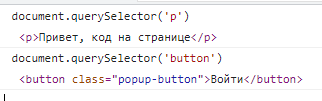


**Clip **

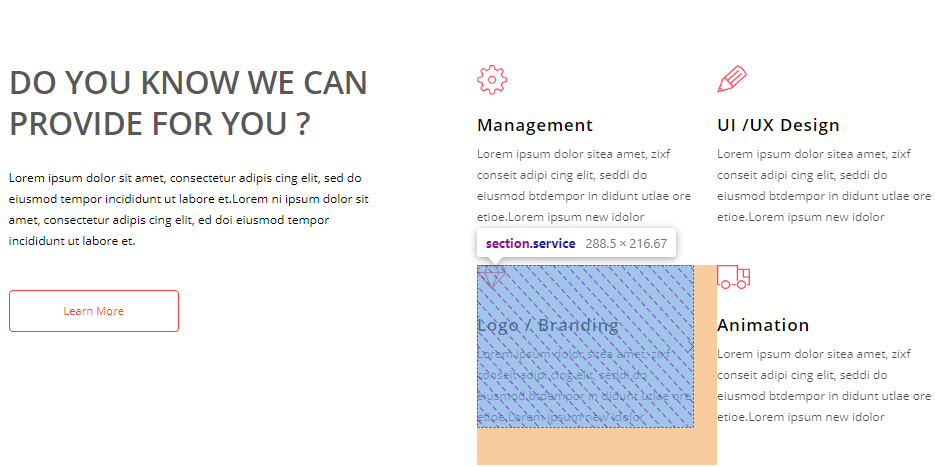
Это свойство действительно задает видимость объекту. Включая его содержимое. Если сделать через CSS чтобы поле изменяло цвет при наведении, но при этом само поле обрезать через **clip.** То наведение будет срабатывать, только когда наведем мышь на необрезанную часть.

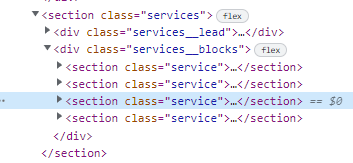
 

**querySelector –** метод выбирает первый найденный элемент удовлетворяющий условию в скобках. В скобках могут быть классы, id, названия тегов.



**Еще чем удобен этот метод, в него можно передавать цепочки последовательностей.**

****

** **

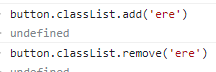
Если нужно найти несколько узлов, то можно использовать **querySelectorAll**



События и обработчики событий. Событие это какое-то действие на стороне клиента. Клик мыши, наведение и тд.

Улавливаем мы события через метод **addEvenListene().** Принимает событие, которое нужно прочитать и функцию, которую нужно применить. Среди события есть например наведение мыши на объект или из объекта. **mouseover, mouseout.**

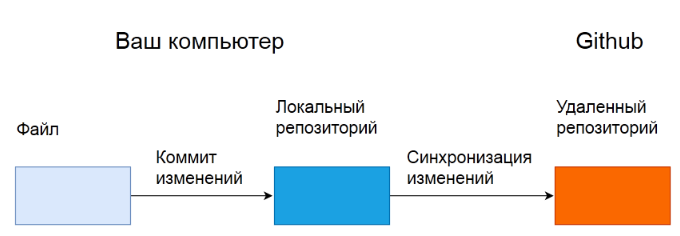
Также помимо **toggle** можно в ручную добавлять и удалять у объекта классы.



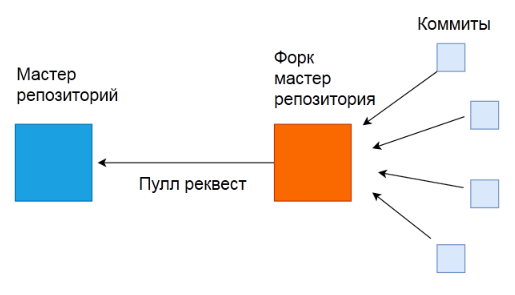
**A4.13 Гит, Github и Github Pages**

**Контроль версий.** Самая популярная **GIT.**

Гит позволяет создать репозиторий, который запоминает все изменения, **коммиты.**

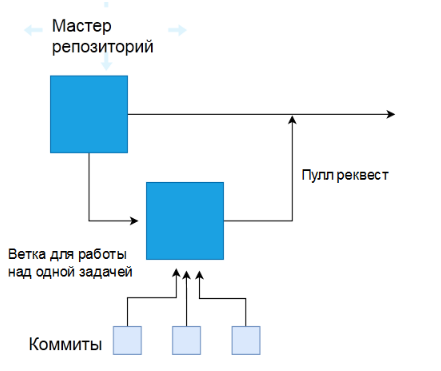


**Можно создавать копии репозитория так называемые форки, чтобы вносить разные изменения.**



Тут просто перед тем, как продолжить работу с другого компа нужно сделать **pull.** Чтобы синхронизировать изменения.

Или работать параллельно, без помех друг-другу. Это называется **ветками.**



Когда работа закончена, ветку или форк можно объединить, слить с основным репозиторием. **Смерджить с мастером.** Для этого нужно сделать на слияние, **pull request,** который ответственный человексможет проверить на наличие ошибок и принять его.

Работать с **git** можно с помощью терминала. Но можно и из редакторов кода.

**Ну… сложно.** Запомнил:

**git add имяФайла**

**git commit –m ‘комментарий коммита’**

**git push**

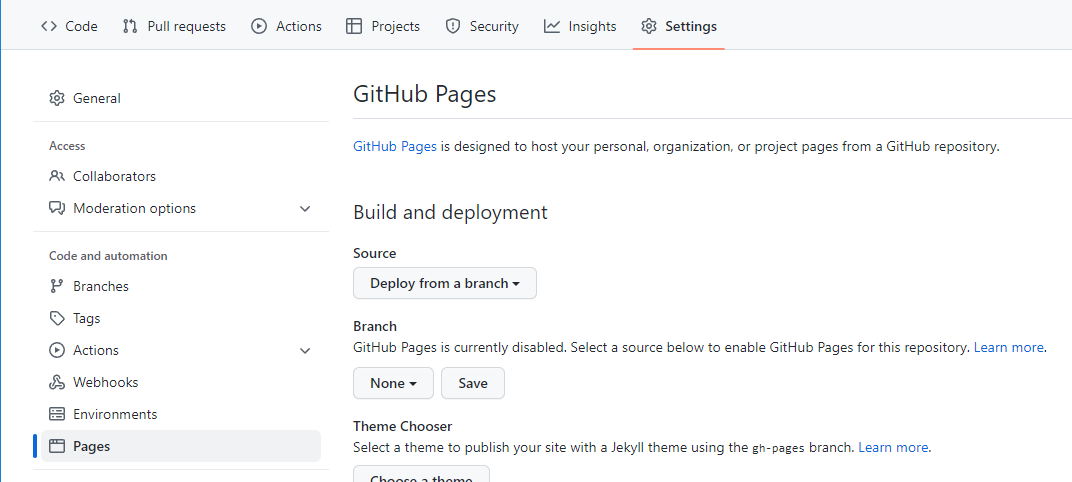
**git status**

**git pull –** синхронизироваться с главной веткой

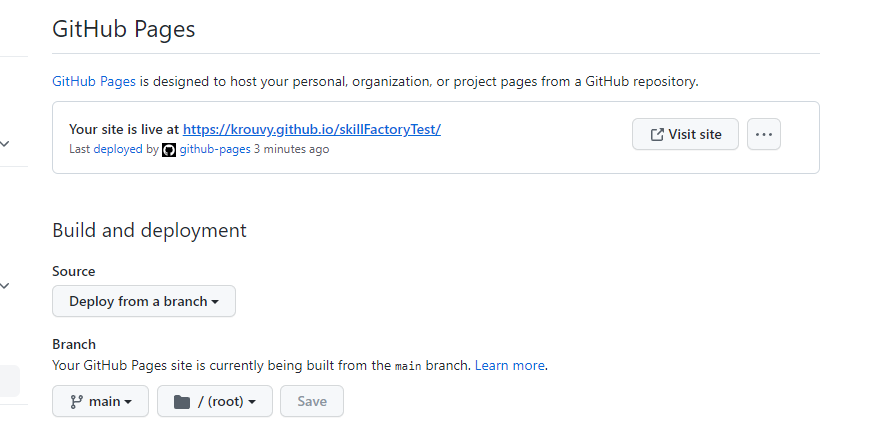
Но через эти команды делать вообще не хочется.

**Github Pages –** Гитхаб позволяет не только хранить ваши проекты в виде репозиториев. Но и использовать его как **хостинг** для фрондэнд проекта. Ну то есть без **backend.**

Для этого нужно перейти в настройки. Найти **GitHub Pages.**

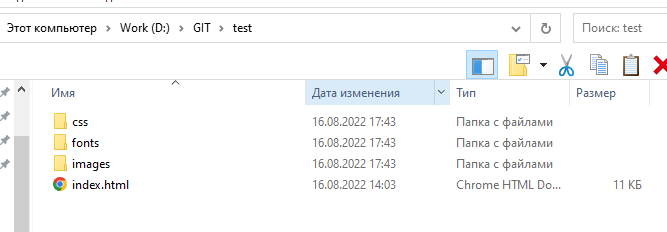


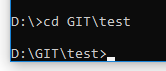
**Но целый проект так опубликуешь. Для этого это должен быть проект в корне которого лежит index.html. В разделе branch нужно выбрать вместо none main и нажать save.**

****

**Видео.** Не знаю буду ли повторять все из видео. Но 1.5 посмотрю хотя бы. Ладно. Я попробую добавить репозиторий как на видео.

Вот у меня есть папка с кодом и файлами.



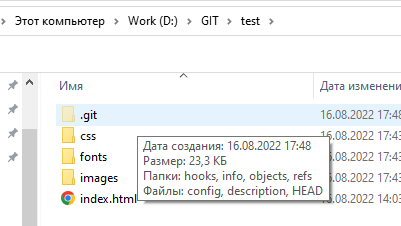


Перейду в эту папку d:

Чтобы сделать из этой папки репозиторий нужно написать:

**git init**

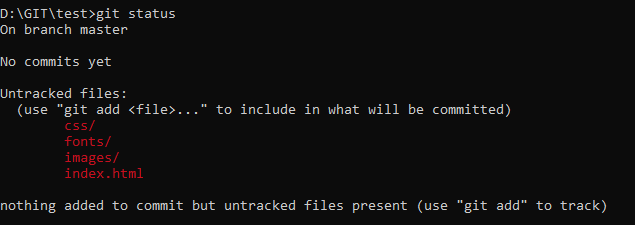
****

****

В папке появляется скрытая папка. Именно она и хранит инфу о всех изменениях.

Узнать информацию о репозитории можно с помощью команды:

**git status**

****

**On branch master** **–** главная ветка(На гитхабе вроде называется **main**)

No commits yet – еще не было коммитов

**Untracked file:** неотслеженные файлы.

Нужно их добавить. Командой **git add**

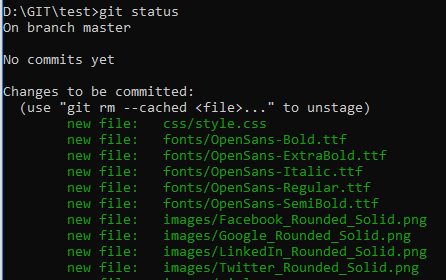
Но можно их добавлять как по одному и писать

**git add название\_файла**

3 раза. Или же поставить точку и добавится весь корень.



Если снова ввести **git status.** То:



Выведен список изменений готовых к коммиту.

Если нужно что-то откатить, то можно ввести

**git rm –cached <file>**

Чтобы сделать сам **commit,** то есть внести изменения нужно использовать команду:

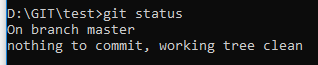
**git commit**

Но если использовать просто ее, то вероятнее всего само сообщение придется писать в каком-то редакторе.

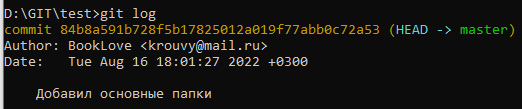
Проще:

**git commit** –m “Сообщение о коммите всегда с двумя кавычками ”

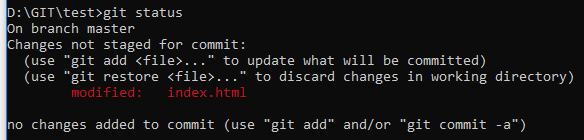
Если сейчас проверить статус, то он будет чист.



С помощью команды **git log** можно просмотреть историю коммитов.



Хорошо. Допустим я внесу изменения в коде в одном из файлов. И снова сделаю **git status**

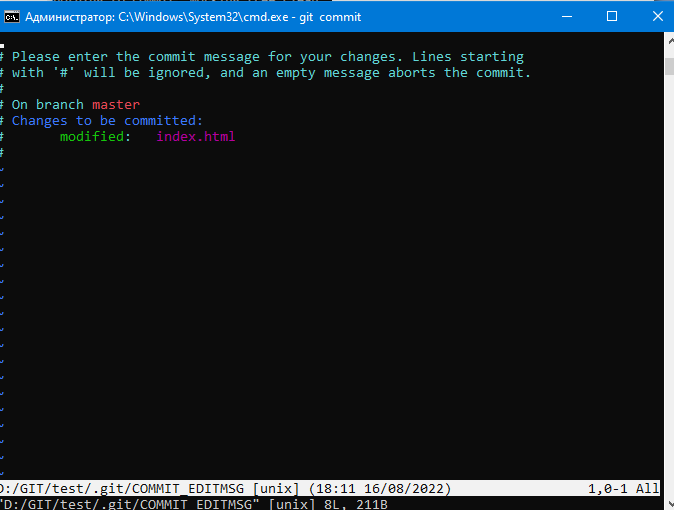
****

Попробую сделать просто

**git add index.html**

**git commit**

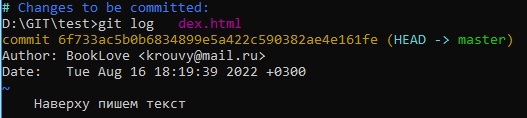
Открылся вот такой редакор. Наверху нужно написать коммент.

****

**Shift + S наверху пишем текст**

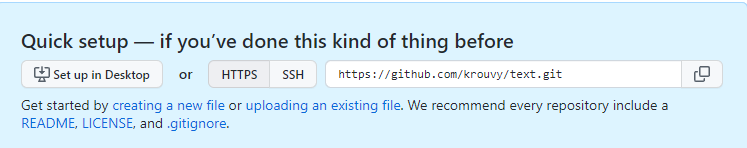
**Ctrl + C**

**Shift + :wq**

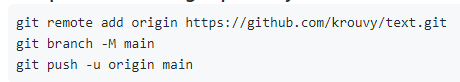
****

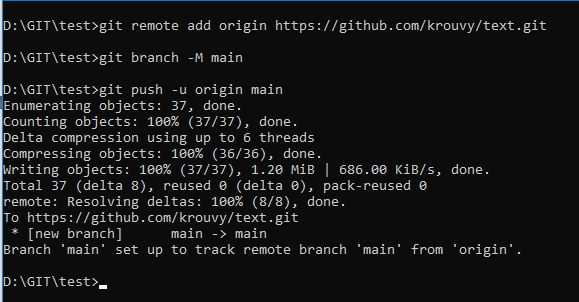
**Ладно. Как добавить его в github?** Сначала нужно создать репозиторий в самом **github.**

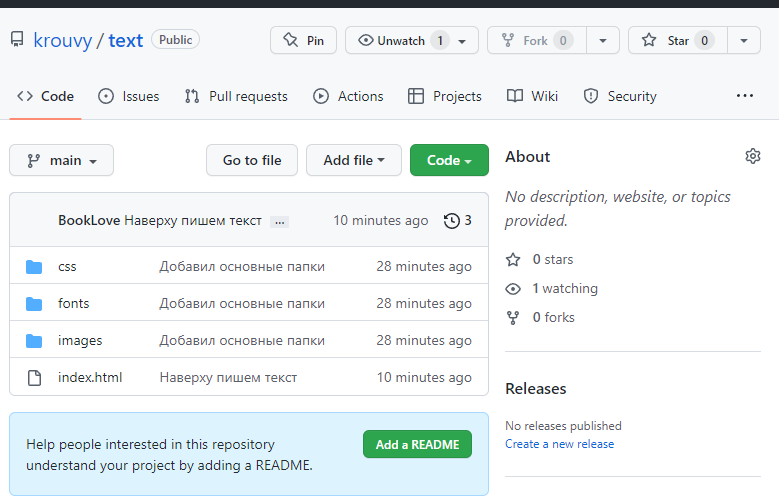
Вот созданный репозиторий. Вот что предлагает сам git



Попробуем отправить сюда существующий репозиторий.

****

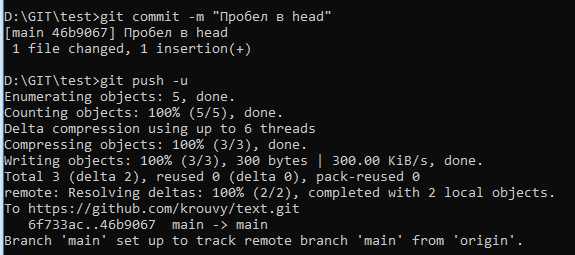
****

****

**Опа. Получилось. Но скорее всего нужно еще права прописать. Или типо того, потому что откуда у git информация, что я имею права в этот репозиторий что-то пушить.**

**Ну во первых перед пушем уже должны были быть сделаны коммиты. И возможны и другие проблемы.**

Ладно. Попробую сделать коммит и пуш.





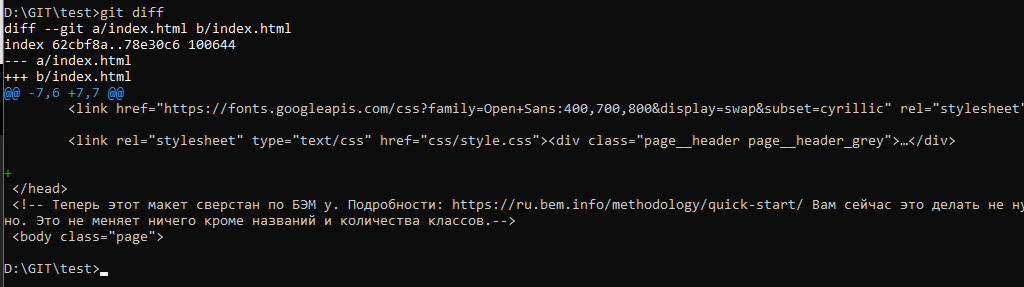
Оно обновляется, просто:

**Git push –u**

Еще одна полезная команда

**Git diff**

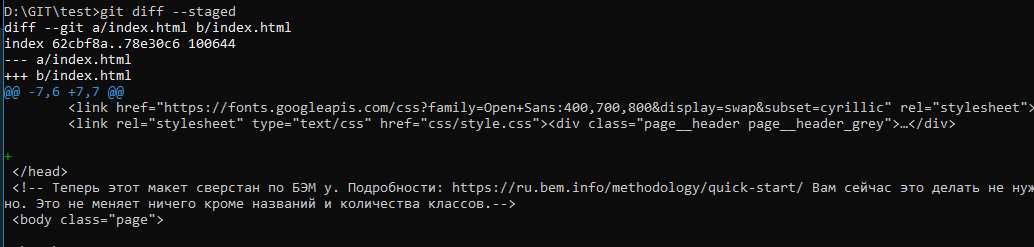
Он показывает изменения которые еще не заккомичены и не добавлены.



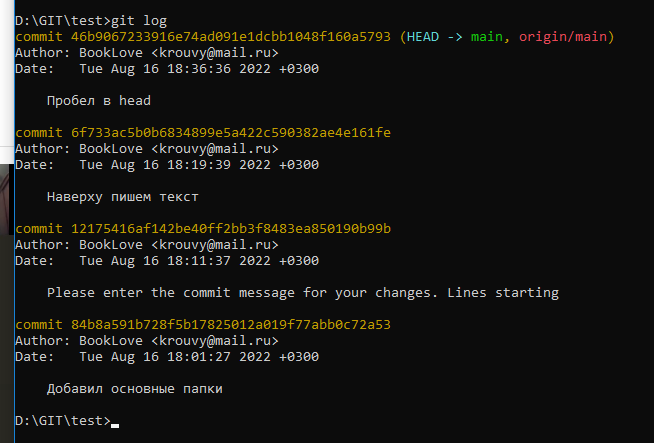
Но стоит сделать **git add** изменение уже не будет.

Если нужно посмотреть уже добавленное изменение, но не закомминченное. То команду:

**Git diff –staged**

****

А сейчас попробуем откатится. Посмотрим все коммиты через **git log.** Скомируем уникальных хэш номер коммита.



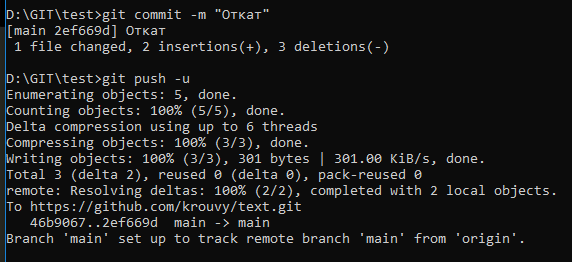
**Git checkout хешномер названиеФайла**

И видно, что файл снова старый: видно что изменился только что



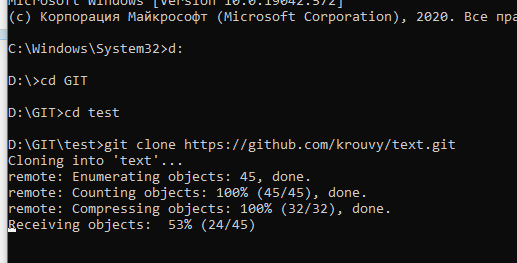






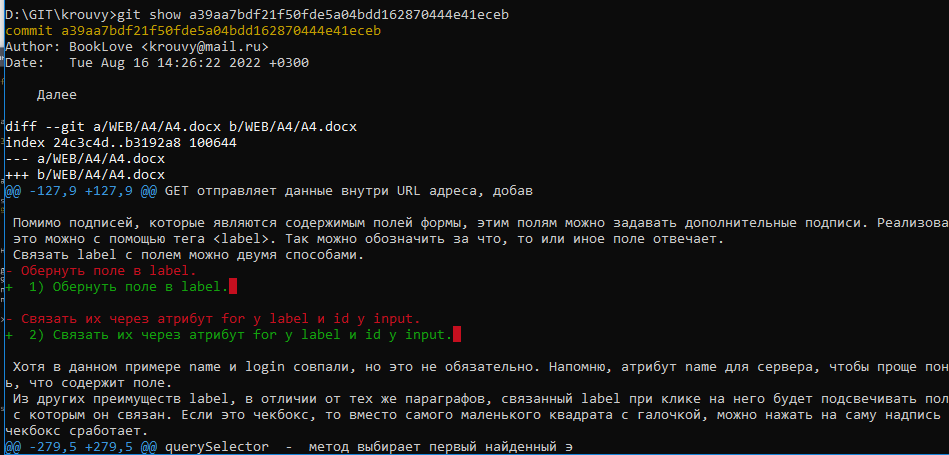
Ладно сложно. Но что насчет клонирования проекта через терминал?

Вот так я пробую клонировать.

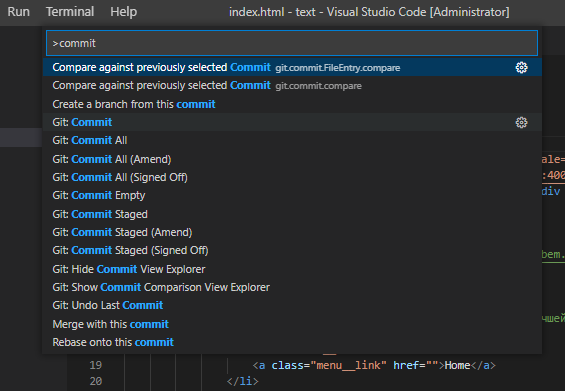


И да.. все пашет. Только я все еще не понял откуда у меня доступ к пушам.

С помощью команды **git show** токен можно посмотреть содержимое этого коммита.



Через ctrl shift p можно открыть команды и например на ctrl alt c назначить git commit



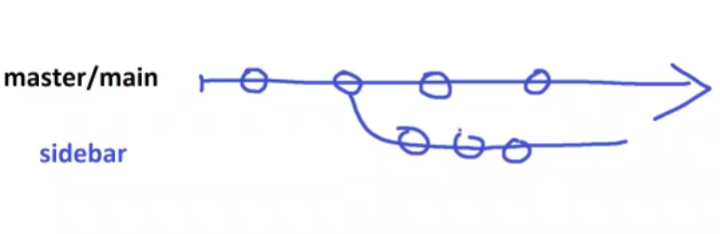
**Pull request.**

**Ветки,**

**Workflow –** рабочий процесс.

Конфликты веток

Есть master/main ветки. И допустим создается отдельная ветка **sidebar** и ее изменения никак не влияют на основную ветку.



И после того, как все протестировано ее можно слить с мастер веткой. И так ветки могут спокойно работать.

И после того, как изменения готовы нужно сделать ПР – **pull request.** Это заявка на слияние веток.

**Видос – разбор модуля.**

В первых. Вот такие поисковики не нужны в реальных проектах. Данные из формы отправляются именно на сервер. А если нужно все же в гугл, то для этого есть специальные АПИ.

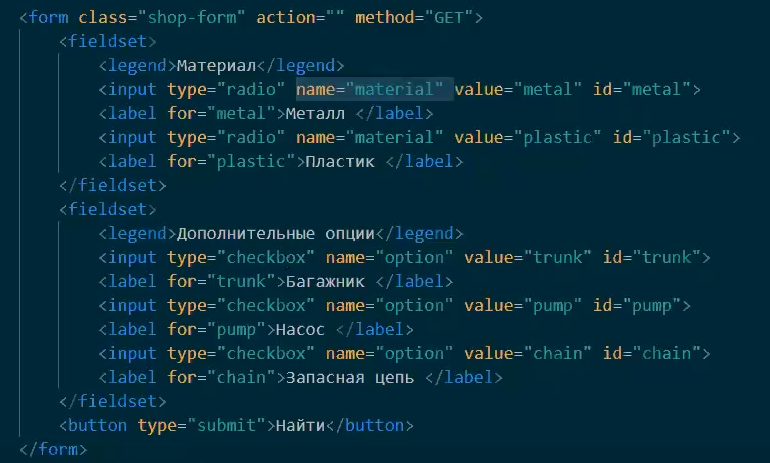


Метод **GET** удобен во время тестирования. Потому что сразу, без какого-либо backend’a виден результат. Метод **GET** отправляет все данные вместе с **URL** адресом.

Например поля с фильтрованными полями будет удобнее отправлять через get, чтобы пользователь видел, что он отправляет. К тому же такой отфильтрованный результат можно сохранить в закладках.

Но например в поле авторизации с паролем, я бы предпочел использовать метод **POST.**

**Input** с текстом не должен содержать **value.** Для этого следует использовать **placeholder.**

****

**Value** для отправки на сервер. **Name** важнее для радиокнопок, чтобы они были связаны и переключались. **Fieldset просто упрощает разделение кода.**

Хотя нет. Name в формах очень нужен. Без него отправка не работает.

У инпутов еще можно задать **паттерн.** Набор правил для ввода.

**Display:none;** работает также как **position: absolute;** То есть они оба вырывают объект из потока и его место занимает тот, что был написан ниже.